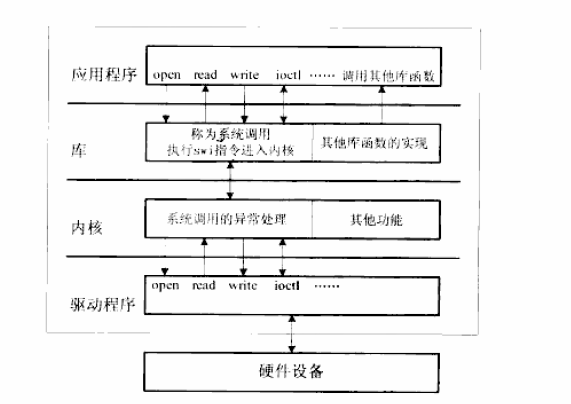
# 驱动的框架

首先我们想写驱动就要先理解驱动是怎么回事

我们使用应用程序的open write 功能的时候其实这个应用层上的open write不是直接操作设备的，而是通过调用库里的函数，通过库传到内核，然后内核找到相应设备驱动程序才操作对应设备、



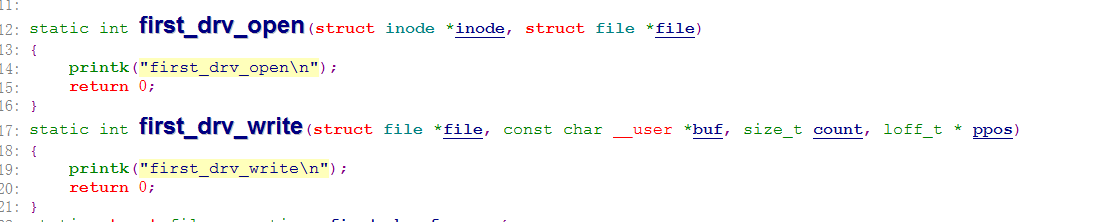
这个图就已经很好的说明了问题，

1. 应用程序调用open函数
2. open传入库参数
3. 库根据参数给内核传入异常指令
4. 内核根据异常指令进行处理
5. 内核根据相应参数对应找到相应设备的驱动程序
6. 对硬件设备进行操作

内核的绝大部分都是由驱动组成的

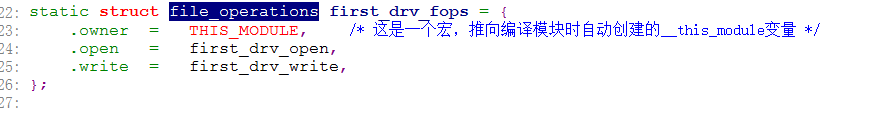
我们可以用代码的简单实现来了解一下驱动的整个框架是怎么回事

首先我们应用层需要用到的功能代码我们要先搞出来，比如说open，write之类的



参数我们可以仿照别的驱动程序中的读取来写

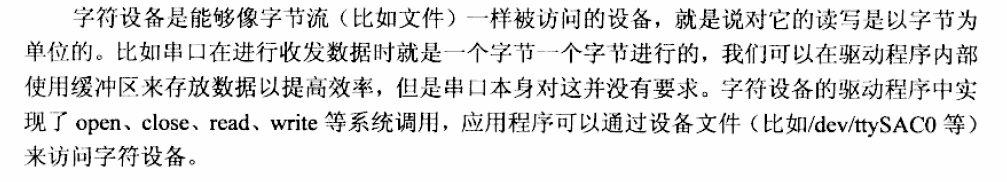
然后我们需要定义一个file\_operations结构体用来存放这些函数



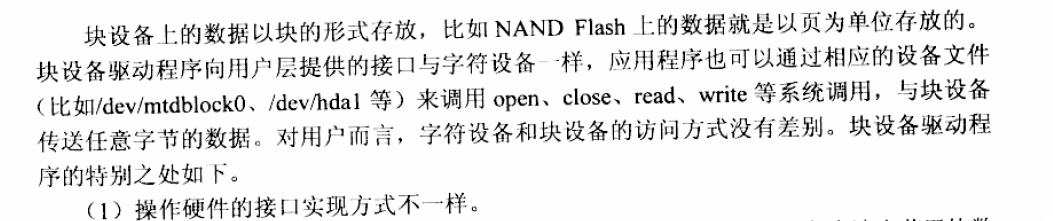
但是光有这些是不行的，我们需要告诉内核我们的这个函数是对哪个设备有用

在这里我们首先要知道一个东西，那就是字符设备怎么看，什么是它的主设备号什么是次设备号

首先我们可以在linux下执行 ls –l /etc一下看一看，那里每个设备前面都有一串字母比如说“crwxrwxrwx”，这个c就是代表的是字符设备文件，后面的我们暂时不用管，后面的是一些权限，字符设备韦东山的书里面有详细说明



b是块设备



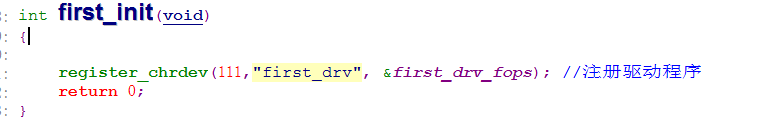


我们需要注意一点，网络接口既不属于字符设备也不是块设备因为网络接口拥有这两个设备的特性

主设备号主要就是说明这个设备是属于哪一类的东西，次设备号就是说这个设备是属于这类设备的哪一个，比如说串口0和串口1 的区别

然后我们继续说代码

我们要想让内核知道这个open函数是作用到哪个设备，我们就得需要告诉这个内核这个结构体的主设备号以及它的名字



register\_chrdev这个函数就是在内核中把这个open之类的操作函数注册到内核中去，

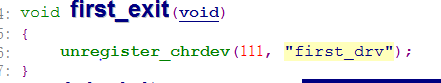
内核中有一个chrdev这样的数组，它分别存放着每个设备的主设备号，然后在主设备号这个空间内存入这个设备的名字，然后再把file\_operations结构体放进去，这样就把驱动程序加载进去了，当我们用的时候我们就先找到主设备号，然后找名字，找到后运行这个结构体中的对应操作函数就可以了，这个数组是存放在内核跟应用程序的接口中：VFS（虚拟文件系统）也就是面向上层的文件接口，我们就是通过它来转换的物理接口

我们继续，这时候内核不知道我们用哪个注册函数，因为可能会有second\_init之类的函数存在，内核不知道用哪个，我们就要加入



因为内核不知道我们用的是哪个注册驱动程序的函数，所以我们要用这个函数定义一个结构体指针来指向这个函数

我们光加载进去了，我们还得有卸载功能





我们添加这两个函数就可以了，原理跟上面是一样的

这个就是加载驱动的整个流程

**mdev的实现以及将驱动加入到内核中**

我们把驱动程序已经写好生成了ko文件，那么我们接下来就要将这个ko文件加入到内核中去

如果只是加载到内核中去我们看不到一个具体的现象，所以我们需要写一个测试函数，这个测试文件在韦东山的例子里面有，我就不说了

我们将那个测试文件编译一下，然后放到我们之前编译好的根文件系统文件夹里，然后我们打开内核，在根文件系统里面输入insmod 文件名.ko就可以了，这时候我们可以lsmod看一下，或者我们cat /proc/devrices 看一下，这时候已经把驱·动文件加载进去了

我们这时候运行测试文件是不行的，因为测试文件里面有这么一句话



我们需要在/dev目录下创建一个字符设备文件xyz，所以我们chmod /dev/xyz c 主设备号 次设备号就可以了，然后我们再运行就可以了

这样很麻烦，我们的linux有个udev机制，就是可以自动创建设备节点，没必要我们自己一个一个地创建，在我们的busybox有一个简化版的udev叫做mdev

这个就是所谓的mdev

我们知道了它是什么东西之后就可以用代码实现它了

我们仿照内核别的代码







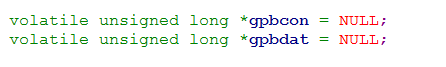
我们加上这几个就可以了，再次编译，卸载上次的first\_drv，rmmod，再次insmod我们就可以看到/dev下有xyz设备节点了

接下来我们来自己写一个led驱动

## 编写led驱动

我们已经把这个框架写完了，我们接下来就需要填充一些代码就可以了

既然我们要写led的驱动，我们就要对寄存器进行操作，首先我们要声明寄存器，因为linux是用的MMU，所以我们需要用虚拟地址来映射物理地址，我们先定义两个寄存器的指针



然后使用下面的函数将地址进行映射



第一个参数是这个寄存器的物理地址，第二个参数是大小，16个字节大小

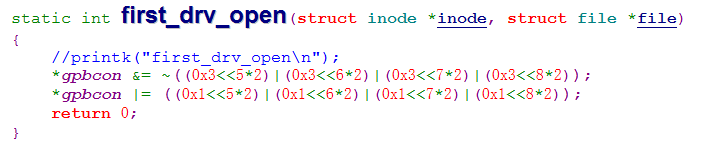
既然init加上了虚拟地址的注册，那么exit也肯定要有关于虚拟地址的卸载



我们就设定open函数是设置led管脚

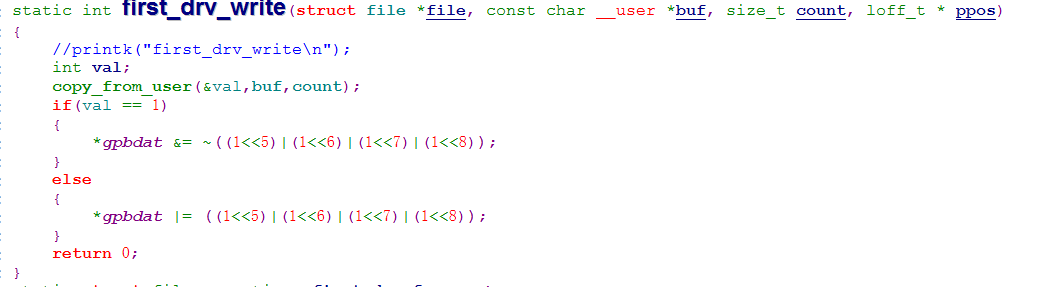
write就执行点灯和灭灯

下面是open函数

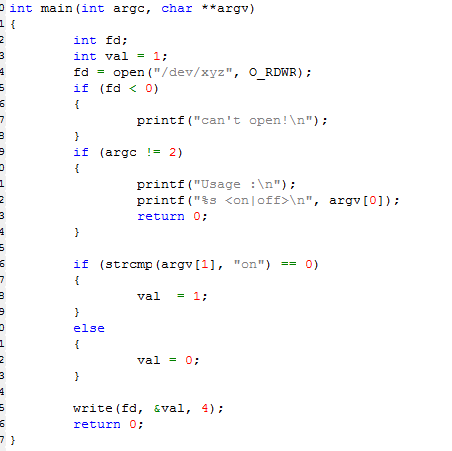


这里没有什么好说的，只不过就是把裸机部分的设置管脚放到这里了

接下来我们看write函数



这里我们需要先看一下测试程序



这里面用到了主函数的argv和argc参数，这代表我们执行这个程序的时候需要输入参数

比如说ledtest是程序，在linux中需要./ledtest是执行这个程序

ledtest这其实就算一个参数，argc其实就是代表的是参数的个数，如果什么都不加的话只是执行./ledtest那么argc就为1，argv[ 0 ]的内容是ledtest

所以我们需要灯亮的话就需要输入./ledtest <on/off> 这样就会检测到什么时候开灯什么时候关灯了

我们根据程序就知道了如果是on的话val等于1 ，off等于0所以write函数就不难理解了，val等于1的时候开灯，等于0 的时候关灯

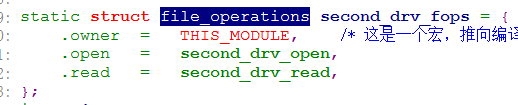
我们也可以利用此设备号单独操作每个灯，韦东山已经解释过了这里就不再赘述，我们可以去看韦东山给的源代码

## 编写按键驱动

我们再写一个按键的驱动，按键的驱动跟led驱动相差不大，我们先把整个驱动的框架建好：初始化函数second\_init和初始化函数的入口，使用的license，还有卸载函数和入口，再来一个file\_operations结构体，还有一个read和open函数就可以了，我们都可以参考上一个led驱动

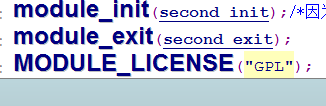








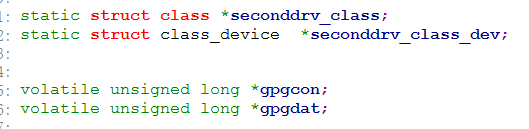




当我们把这几个函数写完的时候我们就已经把整个框架建好了

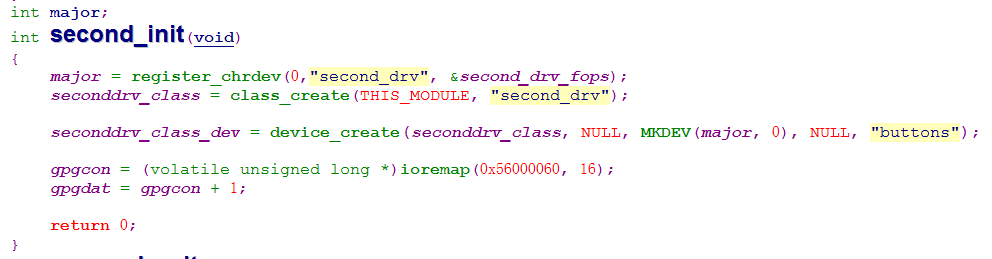
接下来就是填充东西

我们先定义这么几个变量



这里上面两个是创建设备节点的相关变量、下面两个是按键的寄存器的相关变量

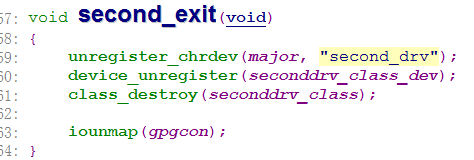
然后我们先去填充初始化函数，这个函数的大致内容跟led驱动的没什么太大的差别



创建设备号以及设备节点和创建虚拟地址

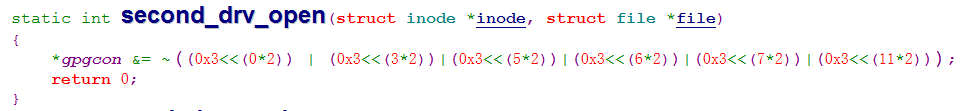
同样的有init就要有exit

exit函数跟led驱动也差不多

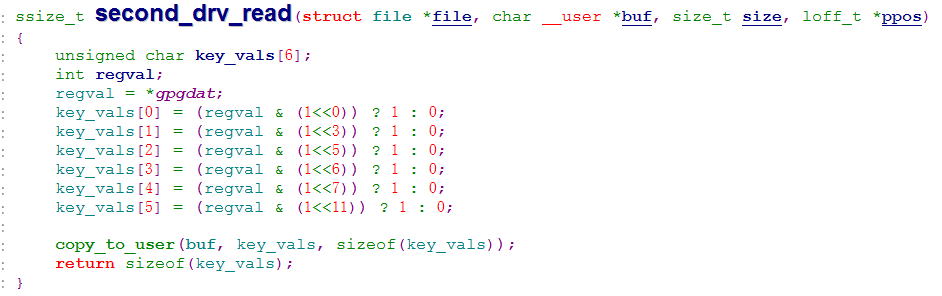


然后我们继续写read和open函数

open函数就是对各个管脚进行设置，设置成输入（也就是设置为零）



read函数其实就是对各个管脚进行的检测，哪个按键按下了



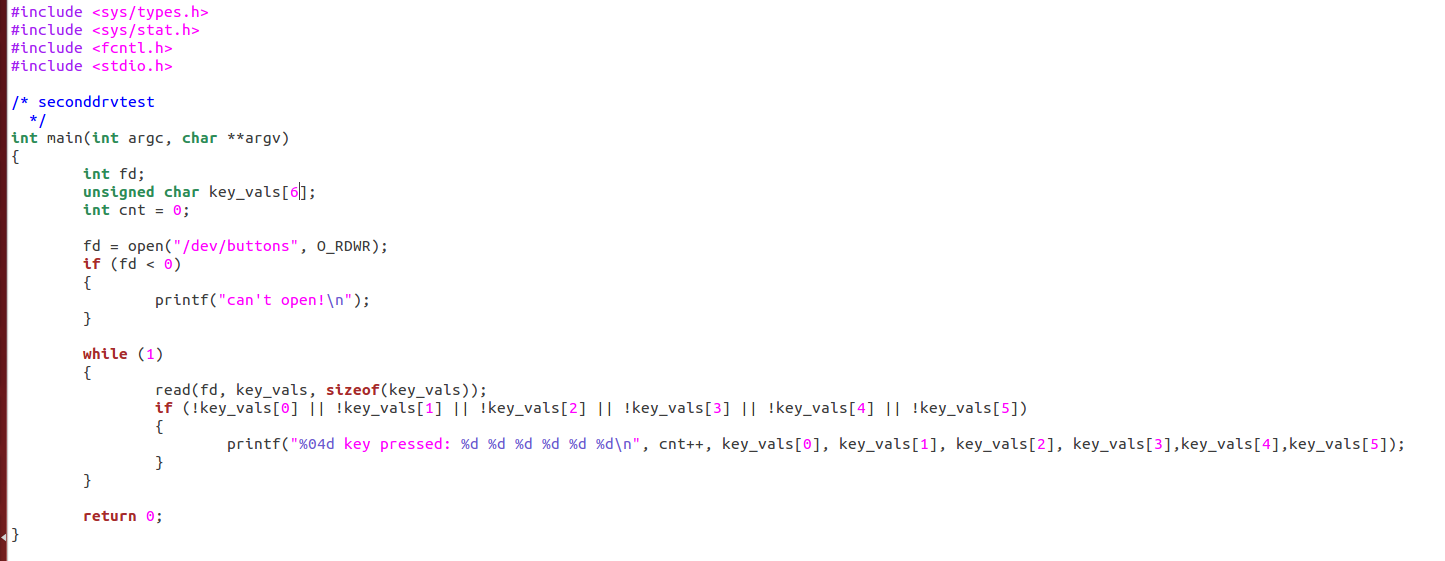
我们定义一个变量来接收GPBDAT的值，然后比看一下哪个值是0，是0就返回零没按返回1

然后用copy\_to\_user返回给应用程序

这样我们的驱动程序就做好了

接下来我们写一个测试程序

测试程序很简单，定义一个数组来接收驱动传过来的数据，然后不断的循环查询之后打印出来



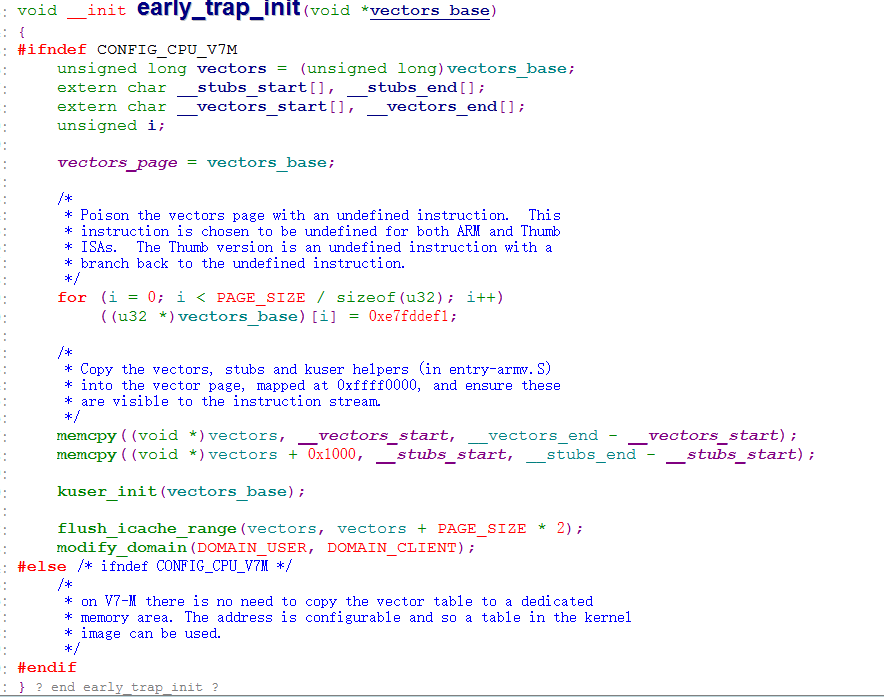
# 内核的异常处理体系

所谓异常就是在CPU进行正常工作的时候，突然被打断了，这种时候这个事件就叫做异常事件，linux内核的异常事件包括：外部中断、未定义的指令、软中断、指令预取终止异常、数据访问终止异常、swi异常等

**中断：**

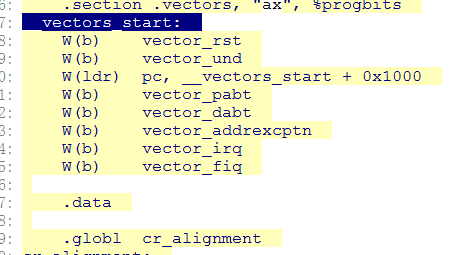
ARM架构的CPU的中断基址有两种0x00000000、0xffff0000,linux内核用的是0xffff0000，其中**early\_trap\_init函数就是要把异常向量复制到0xffff0000中**

**我们来看一下early\_trap\_init**



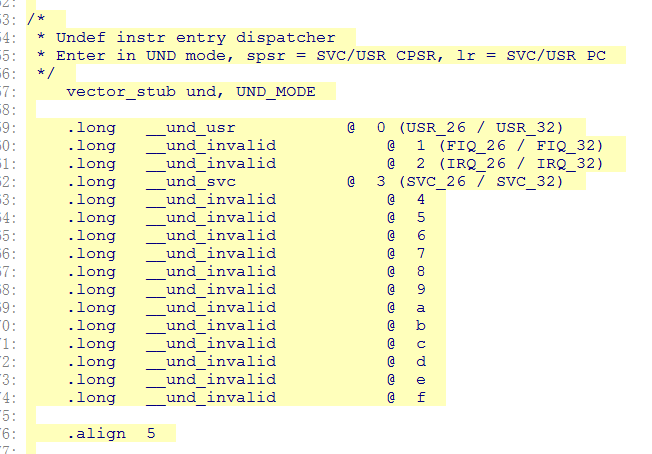
我们可以看到中间两句memcpy，这是将异常向量复制到0xffff0000中去的

我们看一下\_\_vectors\_start，搜索一下

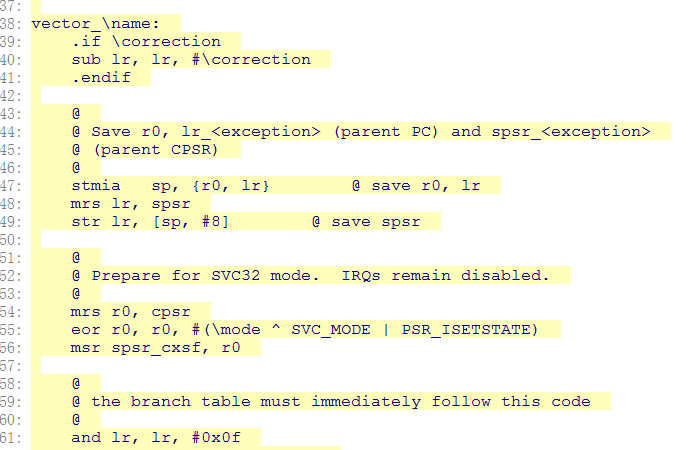


这是在arch/arm/kernel/entry-armv.S中定义的

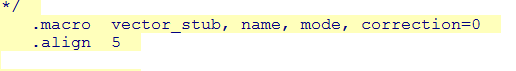
这里面的句子每个都是跳转到相应的函数中，我们以vector\_und为例来看看



vector\_stub是一个宏，我们再去看看这个vector\_stub是什么



这里有个



这个是这个宏定义的格式

name为und mode为UND\_MODE，这里correction没有就默认为0

那么展开之后就是这个

 stmia sp, {r0, lr}  @ save r0, lr  
 mrs lr, spsr  
 str lr, [sp, #8]  @ save spsr

 mrs r0, cpsr  
 eor r0, r0, #(\mode ^ SVC\_MODE | PSR\_ISETSTATE)  
 msr spsr\_cxsf, r0

 and lr, lr, #0x0f  
 THUMB( adr r0, 1f   )  
 THUMB( ldr lr, [r0, lr, lsl #2] )  
 mov r0, sp  
 ARM( ldr lr, [pc, lr, lsl #2] )  
 movs pc, lr ENDPROC(vector\_und)

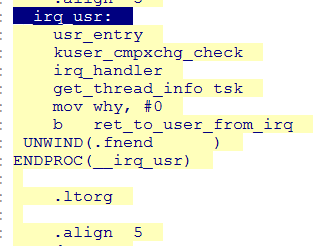
 .align 2  
 1:  
 .endm

这么一个汇编代码，这个就是一些的异常处理函数

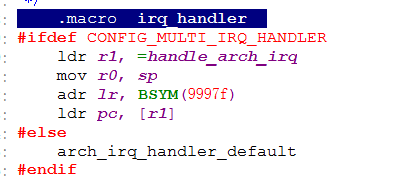
**这里，计算处理完异常后的返回地址，保存一些寄存器，然后进入管理模式，最后根据被中断的工作模式调用上面的某个跳转分支，不同的跳转分支只是在入口处稍有差别，后续的处理大体相同，都是调用相应的c函数，**

我们继续分析，首先我们以\_\_irq\_usr为例看 一下整个调用过程

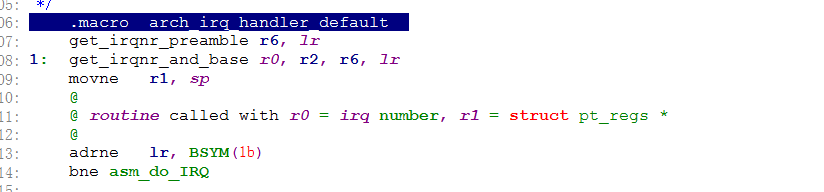
我们先进入这个函数



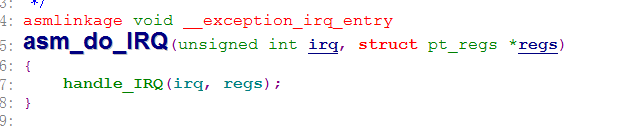
这里有个irq\_handler，我们继续搜索一下



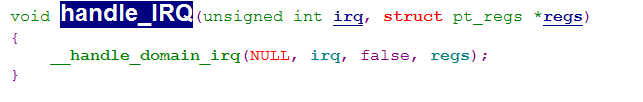
在这里有个arch\_irq\_handler\_default函数，我们继续搜索一下



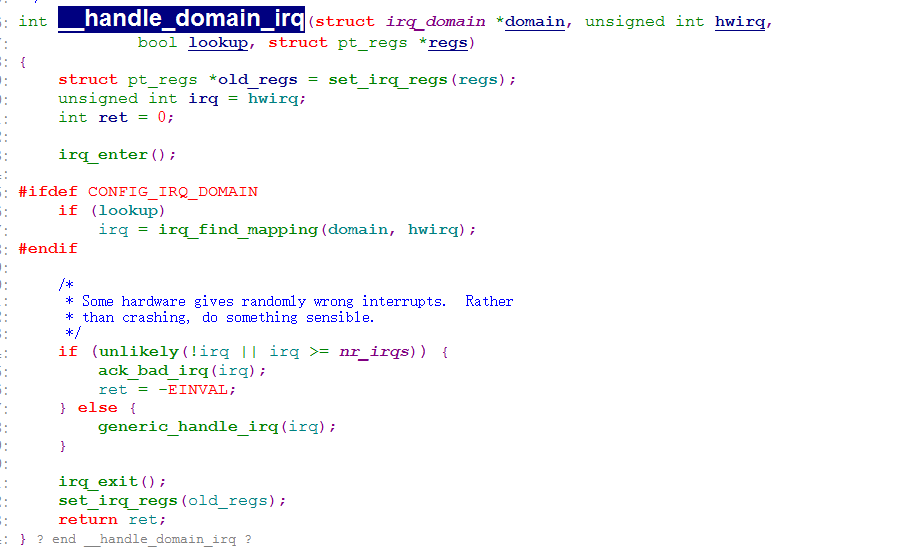
这里就跳转到了asm\_do\_IRQ函数了

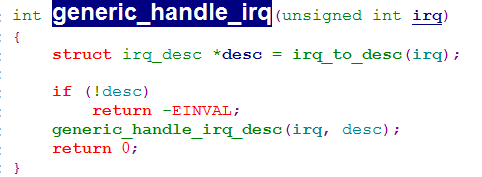


继续跳进去

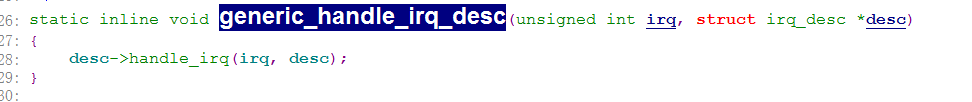


再跳

这里有个generic\_handle\_irq函数，这个函数就是主要的中断处理函数我们点进去

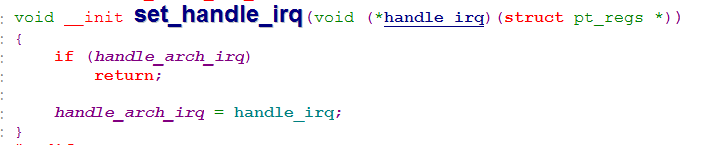


generic\_handle\_irq\_desc点进去



我们搜索一下这个handle\_irq函数

发现在这个函数中调用了

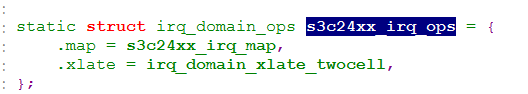


我们再搜索这个函数，看看哪个函数调用它了

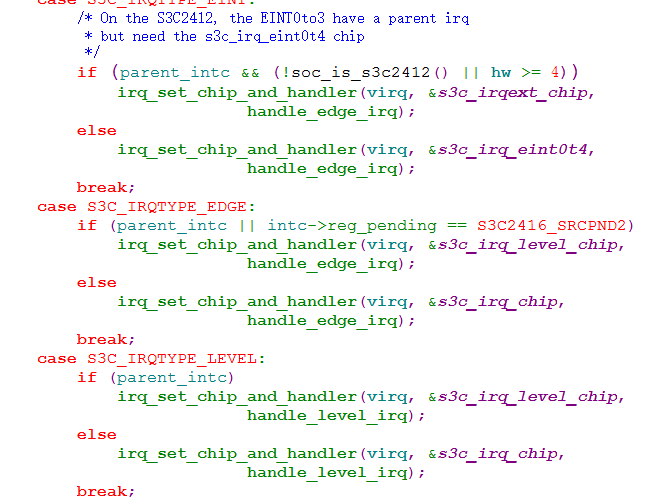
发现是s3c24xx\_init\_intc这个函数调用的它，我们看一下这个函数



我们发现这个函数调用了s3c24xx\_irq\_ops结构体，点进去看看



我们看一下s3c24xx\_irq\_map函数



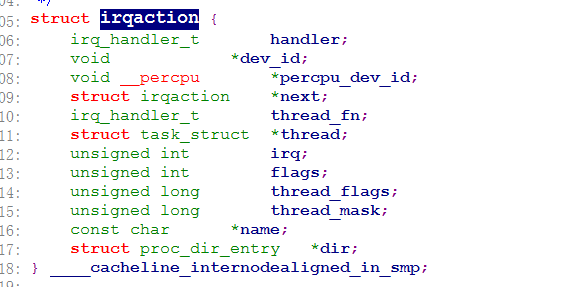
这里有个这个，这是设置一个结构体，我们一开始是有个des\_irq的结构体来描述这些中断，我们定义了一个这个名字的数据结构的数组用数组下标来表示是哪个中断

我们看一下这个结构体



这里就有这个handle\_irq和chip这两个成员，上面这个函数其实就是给这两个成员赋值的，一个是handle\_edge\_irq，另外一个是s3c\_irqext\_chip或者别的中断类型

其中chip是这个芯片的类型的中断类型，我们点开看的话其实是一个结构体装着一堆函数，那些函数就是那个中断类型的中断的处理函数，handle\_irq用来调用这些函数，也可以看成是这些函数的入口，chip用来清除、屏蔽、或者重新使能中断，handle\_irq不仅是入口而且还调用action链表中的中断处理函数



我们用户就是在这个结构体中的handler里面写我们自己的中断处理函数

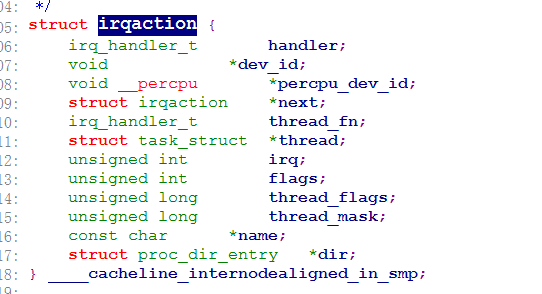
这样整个内核的处理异常过程我们就知道了

1. CPU进入异常模式
2. 调用这些异常函数如\_\_usr\_irq
3. 调用do\_asm\_IRQ
4. 利用irq\_desc结构体调用handle\_irq函数
5. handle\_irq函数调用chip、action链表之类的

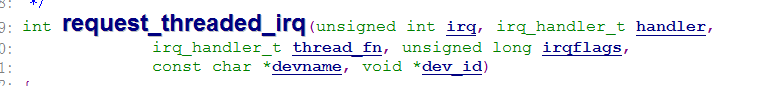
我们再来说一下注册中断驱动程序

我们新的内核注册中断驱动程序的时候使用request\_threaded\_irq函数，我们来分析一下request\_threaded\_irq函数

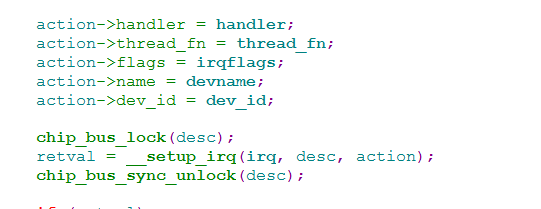
我们上面说到用action链表来放我们用户自己的驱动程序我们先看一下定义action这个链表的类型



可以看出来我们都需要什么参数，我们再看一下这个注册函数的参数



irq是这个中断的中断号，也就是什么类型的中断，handler是我们需要填充的函数，剩下的和前面两个都是构造上面的那个结构体的成员



其中irqflags是代表中断的触发方式

调用\_\_setup\_irq函数用来将这个结构体插入链表

我们说一下\_\_setup\_irq函数的功能

1. 看一下action链表是否为空，如果是就直接插入
2. 看新建的irqaction结构体是否跟链表中的中断方式一致，也就是是否是IRQF\_SHARED，如果是就将新建的链表链入
3. 启动中断，如果irq\_desc[irq]结构中status成员没有被指明IRQ\_NOAUTOEN（表示注册中断时不要使用中断），还要调用 chip->startup或chip->enable来启动中断，所谓启动中断通常就是使用中断。一般情况下，只有那些“可以自动使能的” 中断对应的irq\_desc[irq].status才会被指明为IRQ\_NOAUTOEN，所以，无论哪种情况，执行request\_irq注册中断之 后，这个中断就已经被使能了。

我们总结一下这个函数的功能

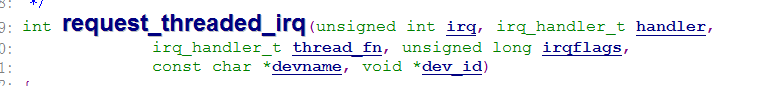
1. 将我们自己写的中断处理函数链入action链表中
2. 确认中断触发方式
3. 使能中断

编写中断按键驱动

我们了解完内核整个的中断体系结构之后我们就可以自己来编写驱动程序了，我们可以参照原来的循环检测按键驱动程序一样，我们可以把整个代码先复制过来，

首先我们先把所有的second改成third，然后我们就可以开始编写驱动程序了

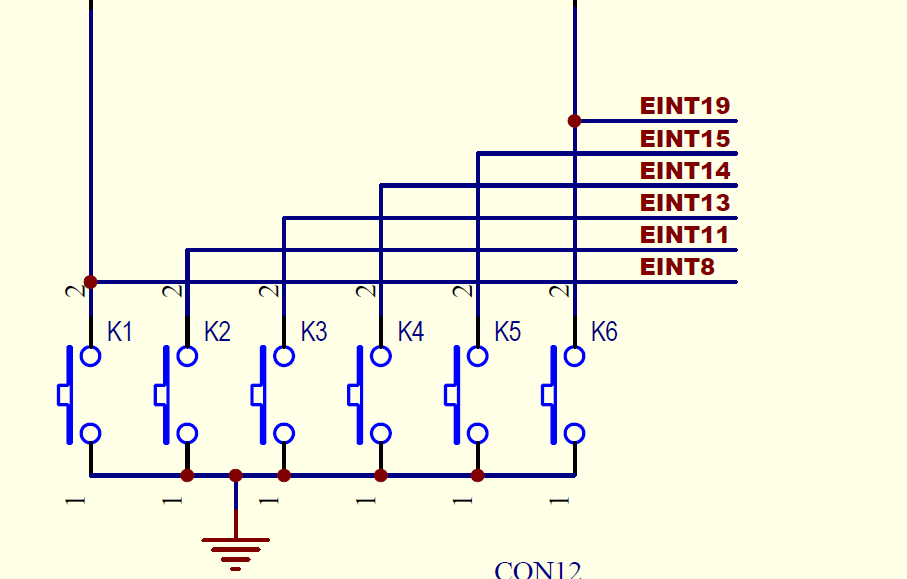
我们先从open函数开始，open函数是做一些准备工作之类的，所以open函数中放着request\_threaded\_irq函数，在以前就说过，这个函数是把整个中断驱动，以及设置中断方式并使能中断，所以这个函数要放在open函数中，接下来我们就要填写这个函数的六个参数，



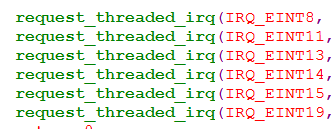
分别是中断号、自己写的处理中断函数、中断进程调度函数、中断方式、设备名字以及设备ID

我们一个一个填写

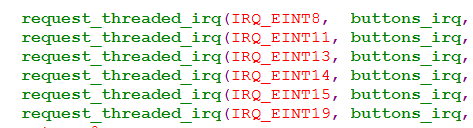
首先是中断号，我们看一下原理图



可以看到我们用的分别是中断8、11、13、14、15、19，我们就填上这些中断方式

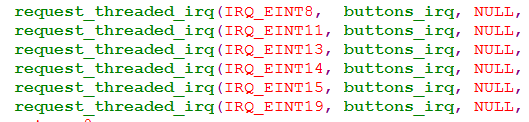


第二个参数是自己写的处理函数，这没什么好说的，下面有处理函数



第三个参数是中断进程调度函数，我们这里没有涉及进程调度，所以我们可以写NULL也可以写一个函数返回IRQ\_HANDLED



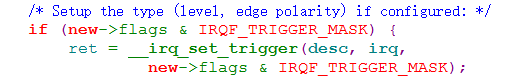


接下来就是设置我们的中断方式，我们这个应该是双边沿触发方式，也就是说不管是高电平还是低电平都会触发，这个明显是设置关于引脚的，我们进入request函数寻找一下这个参数

注意，这里面有一个dev\_id参数，这个参数如果是共享中断就把它设置为0，如果不是可以设置为NULL

我们点进去之后进入set\_irq函数

在set\_irq函数中有段话

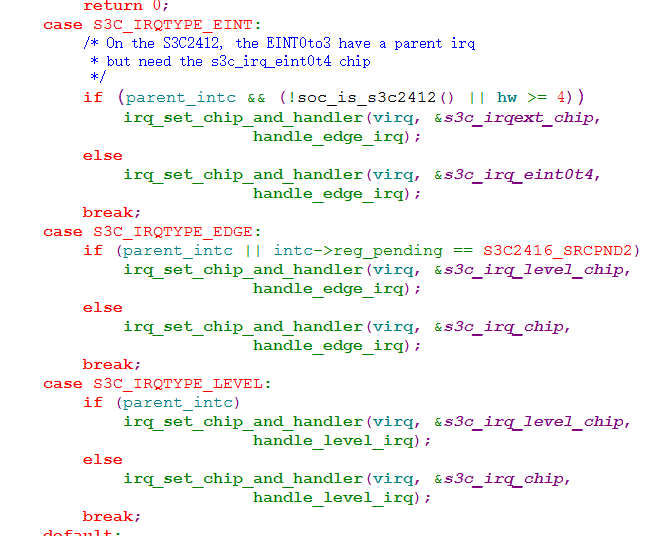


这里有个函数的大致英文意思是设置触发方式，那么应该就是差不多了

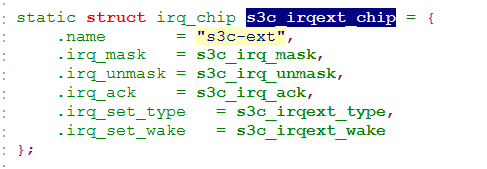
\_\_irq\_set\_trigger，就是这个函数设置

然后我们之前有个设置chip的函数

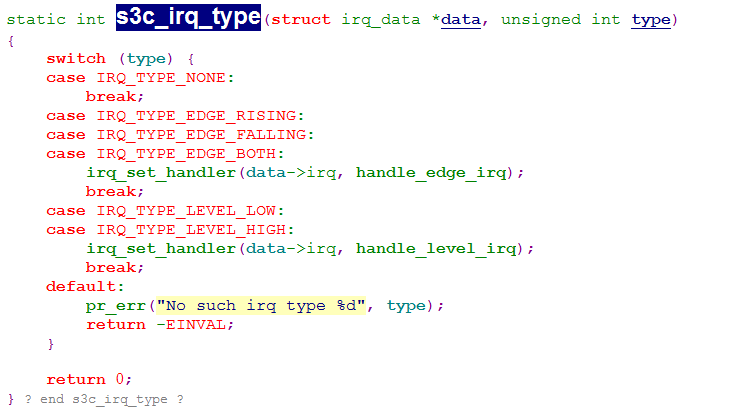
我们看一下



我们随便点开一个看看

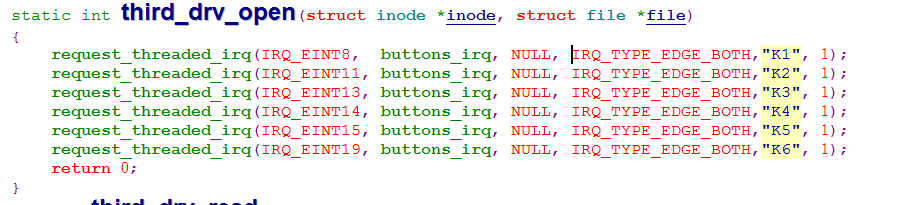


这里有个irq\_set\_type，我们把赋值的那个函数点开看看

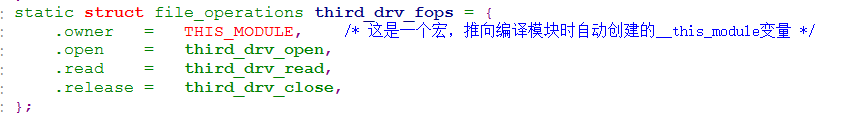


这里面有很多的中断方式，我们需要的是IRQ\_TYPE\_EDGE\_BOTH，双边沿触发

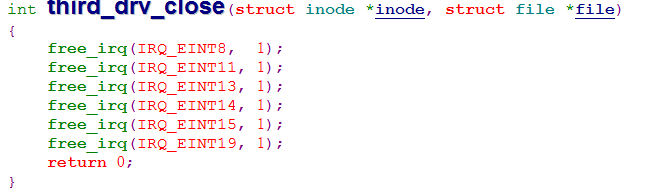
这样我们的request函数的参数的写的差不多了，设备名字我们可以随便起，然后设备ID我们可以都设置成1 ，这个设备ID是为了让我们用一个主设备号下多个次设备号程序分辨准备的，我们这个每个设备都是一个主设备，所以目前为止还不用，所以我们的参数就已经设置好了



既然我们有插入链表的功能了，那么我们也需要有释放这个中断设备的功能，这里需要在我们的结构体里面再加上一个函数

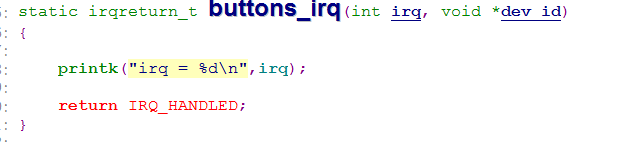


Free\_irq函数就可以释放这个中断



这个free函数只需要它的中断号以及设备ID就可以了

我们现在还差一个处理程序，这个处理程序我们暂时先把它写成没有什么功能，就是打印一下设备号就可以了



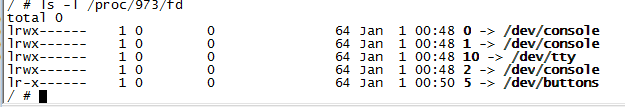
它这里的中断号和ID都是从request函数中得来的，这样我们的一个简单的中断驱动就写好了，我们这个目前还不要测试程序就可以了，我们编译一下之后将驱动打进内核中，我们看一下cat /proc/interrupt，这时候是看不到我们的驱动的，我们需要借助一个指令才可以

exec 我们让它挂到5号上面去 exec 5</dev/buttons，

我们可以ps一下首先看看sh程序运行的进程号

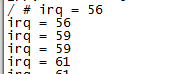
，我这里是973，所以我们查看一下973的fd

ls -l /proc/973/fd一下



我们可以看到buttons的驱动程序已经挂载到了5号上面去了

然后我们再看就会有了，这时候我们按键就会有反应了

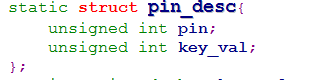


我们不用的时候也需要exec来解除一下

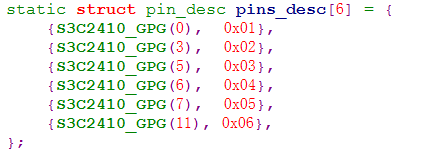
exec 5<&-，这里解释一下，不一定是要5，只要是没有别的设备使用的号码就是可以的

我们现在只是让button\_irq函数打印了一条信息而已，我们现在来填充这个函数

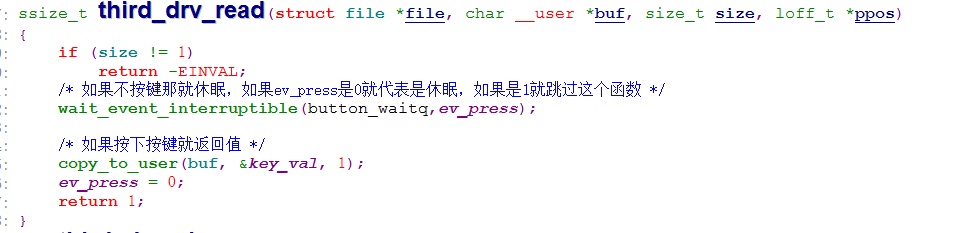
我们让按下第一个显示0x81，松开显示0x01，以此类推，一共六个，我们首先定义一个结构体，里面成员分别是芯片管脚和数值



我们来定义一个这个类型的数组来装我们6个按键的信息

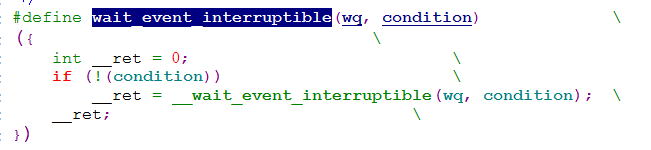


因为我们要写一个测试程序，test，需要像那边传数据，所以我们用到了read函数



因为我们以前用的循环检测的方法十分的耗费处理器资源，我们有一个方法，就是不触发中断的时候是睡眠的，如果触发中断就从睡眠中被叫醒

我们用wait\_event\_interruptible函数来实现睡眠的功能，我们看一下这个函数是怎么写的



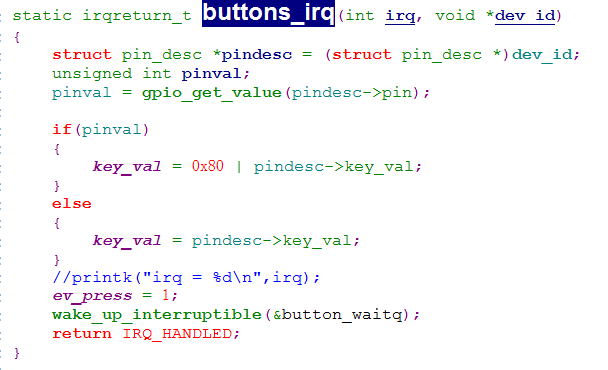
这里已经很清楚了，也就是说当第二个形参给0的时候进入睡眠，给1就什么都不干

这样我们就定义一个全局变量

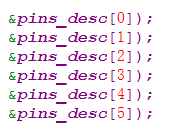


定义了这个之后，我们在read函数中如果没按下键就表示卡在wait\_event\_interruptible函数不动，一旦按下那就程序走下来了，走到copy\_to\_user函数给上层传递参数

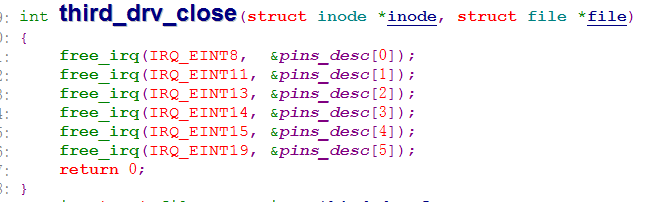
ev\_press变量是在buttons\_irq函数中置1的。我们看一下buttons\_irq函数



这里用函数来获取该管脚的高低电平，，然后进行判断是按下还是松开，然后ev\_press置1

我们把request函数的最后的dev\_id改成

我们把free那里也改成该数组



这样我们的驱动程序大致就完成了，但是我们还需要加一个头文件，因为我们有些用到的函数我们原来包含的头文件是没有的

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/delay.h>

#include <linux/irq.h>

#include <linux/interrupt.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/gpio.h>

#include <asm/io.h>

#include <linux/module.h>

#include <linux/device.h>

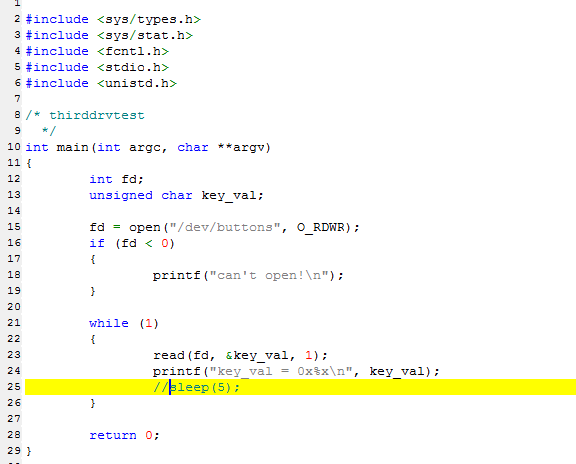
#include <asm/irq.h>

#include <mach/gpio-samsung.h>

#include <linux/sched.h>

这样我们就可以编译过去了

我们再写一个测试程序



然后我们在Ubuntu下编译并拷贝到我们的文件系统中去

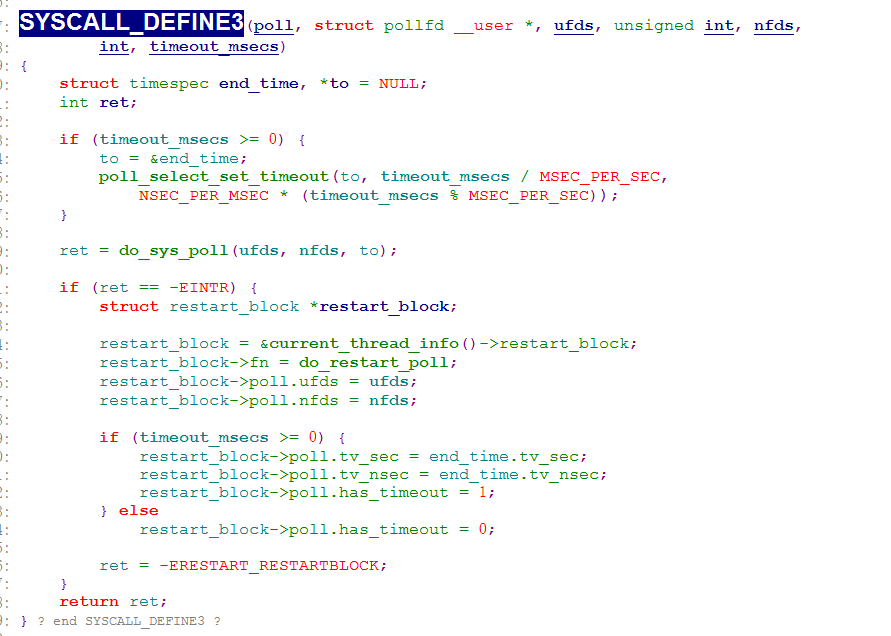
我们装载驱动后不用exec 然后./thirddrvtest &，这样就能让测试程序后台运行，然后我们按键就可以有按键输出了

## Poll机制

当我们用中断来写按键程序的时候是让它休眠，当有按键按下的时候才会去唤醒，我们有一个poll机制，是我们固定一个时间，如果这个时间内不发生中断那也会返回东西，如果按键中断触发了，那就会正常返回按键值

这个poll机制我们来按照代码来讲解

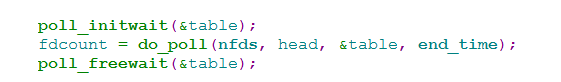
我们应用程序调用函数的时候往往会调用到内核中的函数，比如说read就会调用到sys\_read函数，open就会调用到sys\_open函数，那么我们的poll也就会调用到sys\_poll函数，但是我们的linux-3.18.26用的不是sys\_poll，而是SYSCALL\_DEFINE3函数来代替sys\_poll，我们看一下SYSCALL\_DEFINE3函数



我们可以看到这里有一个ret = do\_sys\_poll(ufds, nfds, to);的调用

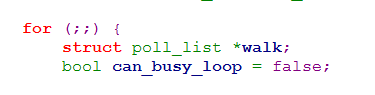
我们点开do\_sys\_poll函数看看

其中有很多我们先暂时不用管，我们看到有一个



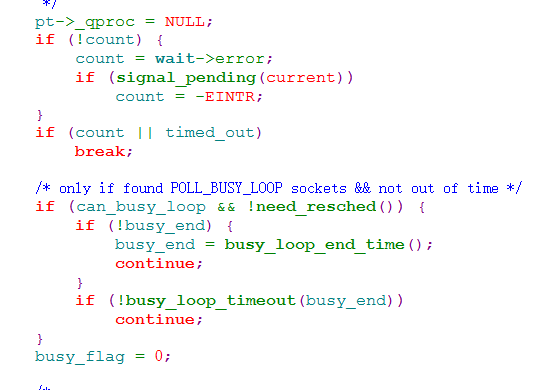
调用了do\_poll函数

点开看看

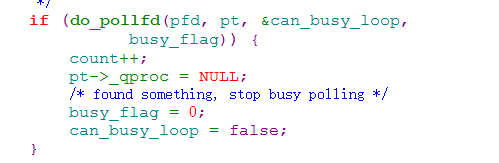


我们这里有个循环

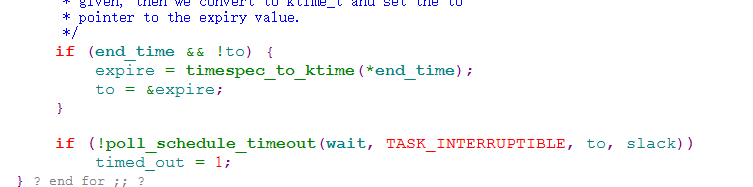
退出这个循环的条件是



count非0



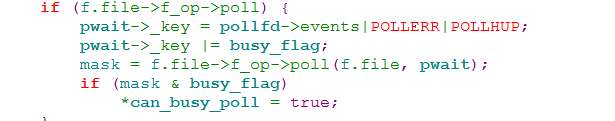
或者说上面的do\_pollfd成功了



这里是让这个程序休眠

只要是应用程序调用poll并且上面的几个条件不满足的话程序就会进入休眠

它可以被驱动唤醒



这是调用我们在驱动里面写的poll函数

我们在驱动程序里面挂的poll\_wait函数只是把我们的驱动程序这个进程拉到等待队列里面去，仅此而已，将程序休眠的是我们之前的do\_poll程序

## 异步通信

我们现在写的按键驱动，不管是查询方式还是中断方式还是poll机制方式都是从应用程序操纵驱动程序

我们的这个异步通信是由驱动程序通知应用程序，也就是说当设备准备好了之后，通知应用程序

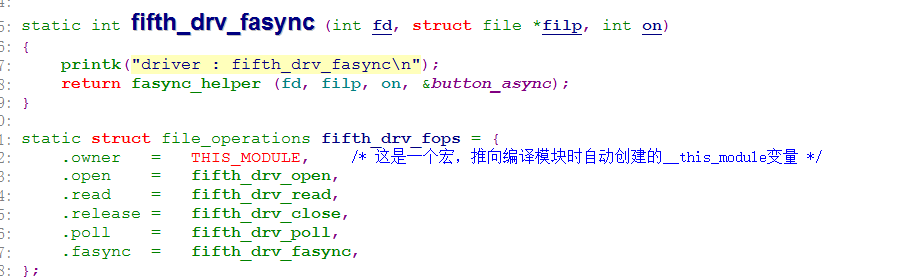
异步通知主要有以下几点

①应用程序：注册信号处理函数

②谁发：驱动程序

③发给谁 => 应用程序 => app要告诉驱动PID

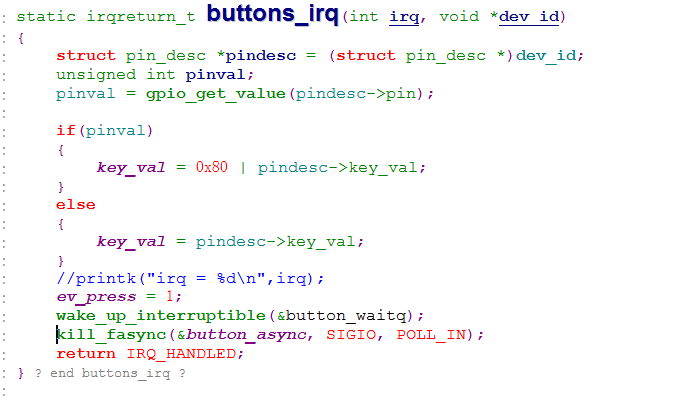
④怎么发：kill\_fasync

我们用kill\_fasync来给应用程序发送通知，我们

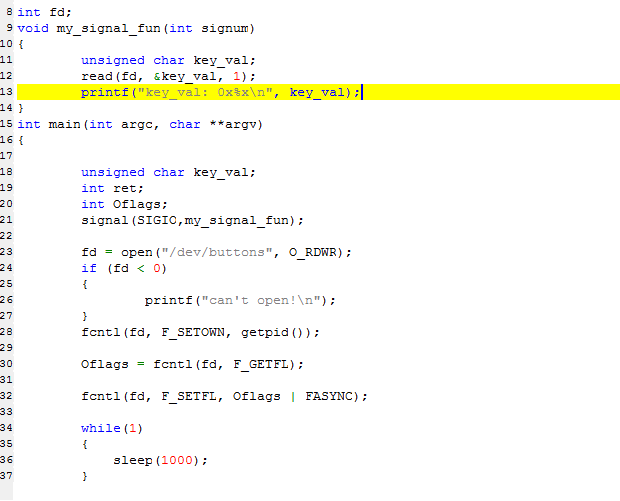
用这个函数来注册并初始化异步通知，然后要定义一个结构体



在



中告诉应用程序，我们应用程序需要把PID告诉驱动，用



fcntl(fd, F\_SETOWN, getpid());

Oflags = fcntl(fd, F\_GETFL);

fcntl(fd, F\_SETFL, Oflags | FASYNC);

告诉驱动程序，用signal打开异步通信

## 同步互斥阻塞

当我们用open函数的时候有可能是会有多个程序都在用这个open函数，因为我们的linux系统是有多进程这么一说的，为了防止这种情况的发生，我们有这几种办法来避免，首先我们可以定义一个变量，在驱动里面定义一个canopen

然后在open函数中写上

if(--canopen != 0)

{

canopen++;

return –EBUSY;

}

然后在close函数中加上canopen++就可以了，这样达到了一个驱动在同一时刻只能被一个程序使用，因为我们运行到open函数中的时候首先canopen要自减1，这时候canopen已经是0了，但是canopen等于0，所以if不成立可以继续运行，然后在close再加回来就可以了，但是如果在这个时候另外一个程序调用了这个驱动的话，canopen已经是0了，然后还要自减一遍，就成了-1，这是候open中的if成立返回-EBUSY报错并canopen加回去

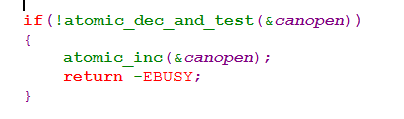
但是这样写还是有漏洞的，因为如果程序运行到了if中，在这个时候突然另外一个程序再运行open的时候还是会打断，所以我们需要在if中一次完成自减和自加的工作

这就需要内核中自带的原子变量了

我们首先定义一个原子变量



然后我们在open函数中做处理



atomic\_dec\_and\_test是自减操作后测试其是否为0，为0则返回true，否则返回false

这样就不会打断我们的操作了，然后在close中加上

就可以了

**我们看一下互斥**

互斥的意思其实就是我们在调用驱动的时候会申请一个信号量，直到这个程序不再使用这个驱动的时候才会释放这个信号量，如果这时候另外一个程序使用了这个驱动的话，就会再次申请这个信号量，但是这个时候上个程序还在使用这个驱动，所以这个程序申请不到信号量，就会进入休眠状态，直到上个程序不用了释放了信号量的时候才会唤醒，这样的好处是再多的程序用这个驱动都可以同时打开

我们改一下这个程序

首先我们需要定义一下互斥锁



这里原来的定义互斥锁是DECLARE\_MUTEX，但是到了现在最新版的内核之后就变成了这个



Down是获得信号量

在close函数中释放信号量



就可以了

**我们再看一下阻塞**

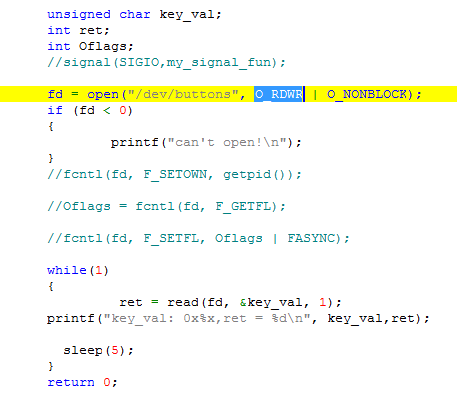
这里分非组塞和阻塞，

非阻塞就是这个程序不进行对设备的操作的时候并不休眠，而是再次循环操直到有返回为止

阻塞就是如果没有满足的条件就挂起进程，进入休眠，当有操作的时候就会唤醒

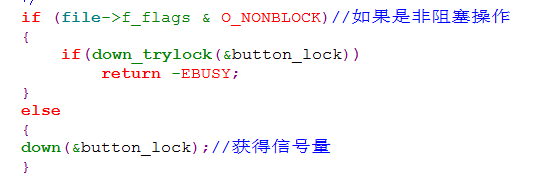
我们改一下程序

我们先改一下应用程序



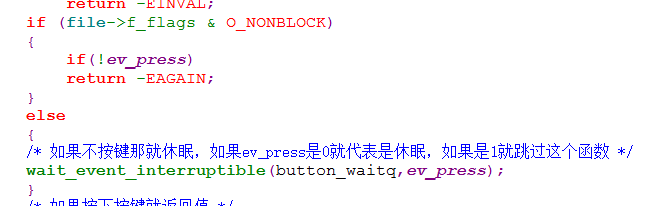
O\_NONBLOCK表示非阻塞，如果没有与上这个就代表阻塞

驱动程序：



在open函数中添加这几个句子，如果是非阻塞操作就获取信号量如果没获取到就报错

在read中加入



如果非阻塞再检测是否有按键按下，如果没有就重新检测

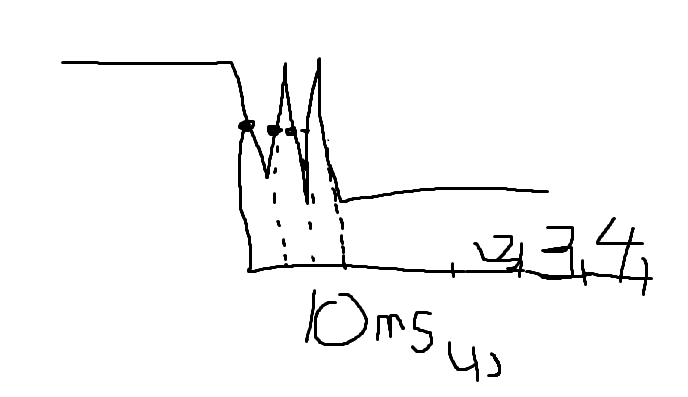
## 用定时器对按键的防抖动

我们按键的时候有时候有肯能会被检测到好几个返回值，因为在按键的时候会有硬件上的脉冲不稳，在我们写单片机程序的时候也出现过这种情况，也就是需要我们说的消抖

我们在写单片机的时候也就是写一个简单的延时，然后再去确定一下是否还是低电平就可以了

我们这回是用定时器弄，我们按键的脉冲是一个一个的尖的波，我们如果在每一个“毛刺”的时候用同一个定时器开始延时

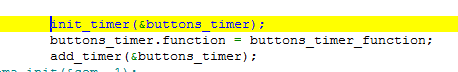
比如说延时10毫秒我们可以延时，然后到另外一个毛刺的时候将定时器的时间



这个图也说明了，也就是说当碰到另外一个跳变的时候这个定时器就被重新装载了10ms，也就是说直到最后一个跳变才会结束，所以这样就可以避免多次跳变

我们看一下代码

首先在我们的init函数中加上



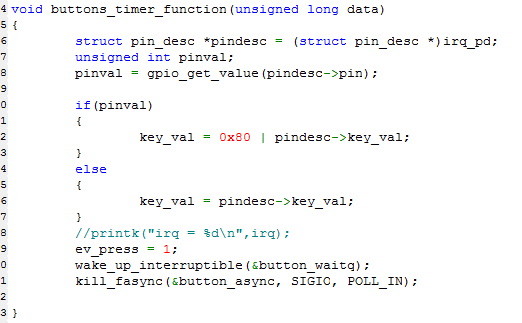
我们再定义一个定时器的结构体



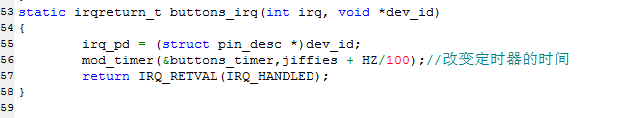
这个结构体里面包含了函数

我们的第二句话就可以看出来那个是定时器的函数add\_timer就是使用timer函数

我们再看一下buttons\_timer\_function函数其实就是原来的中断处理函数，我们把原来的中断处理函数剪切到那里



我们原来的中断处理函数也是需要改变的



我们只要一有中断就重新给定时器一个时间让它重新计时，这样就是达到了上面说的那种效果，测试程序不用变